LISP AND SYMBOLIC COMPUTATION: An International Journal, 10, 223–236, 1997
© 1997 Kluwer Academic Publishers – Manufactured in The Netherlands

Threads yield continuations[*]

SANJEEV KUMAR (*skumar@cs.princeton.edu*)
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

CARL BRUGGEMAN (*bruggema@cse.uta.edu*)
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019-0015 USA

R. KENT DYBVIG (*dyb@cs.indiana.edu*)
Computer Science Department
Indiana University
Bloomington, IN 47405 USA

(Received:)

(Revised:)

Keywords: Concurrency, Continuations, Control delimiters, Control operators, Threads

Abstract. Just as a traditional continuation represents the rest of a computation from a given point in the computation, a subcontinuation represents the rest of a *subcomputation* from a given point in the *subcomputation*. Subcontinuations are more expressive than traditional continuations and have been shown to be useful for controlling tree-structured concurrency, yet they have previously been implemented only on uniprocessors. This article describes a concurrent implementation of one-shot subcontinuations. Like one-shot continuations, one-shot subcontinuations are first-class but may be invoked at most once, a restriction obeyed by nearly all programs that use continuations. The techniques used to implement one-shot subcontinuations may be applied directly to other one-shot continuation mechanisms and may be generalized to support multi-shot continuations as well. A novel feature of the implementation is that continuations are implemented in terms of threads. Because the implementation model does not rely upon any special language features or compilation techniques, the model is applicable to any language or language implementation that supports a small set of thread primitives.

1. Introduction

Continuations have proven useful for implementing a variety of control structures, such as nonlocal exits, exceptions, nonblind backtracking [28],

[*]This material is based on work supported in part by the National Science Foundation under grant numbers CDA-9312614 and CDA 93-03189.

nondeterministic computations [8, 14], coroutines [12], and multitasking [7, 15, 30], at the source level. Subcontinuations are more expressive than traditional continuations and may be used to implement similar control structures in the presence of tree-structured concurrency [16, 17]. This article describes a thread-based implementation of one-shot subcontinuations that has been incorporated into a multithreaded implementation of *Chez Scheme* on an SGI *Power Challenge* multiprocessor.

Just as a traditional continuation represents the rest of a computation from a given point in the computation, a subcontinuation represents the rest of a *subcomputation* from a given point in the *subcomputation*. The base of a subcomputation is specified explicitly, and a subcontinuation of the subcomputation is rooted at that base. In contrast, a traditional continuation is rooted implicitly at the base of an entire computation. Subcontinuations may be used to implement traditional continuations by introducing an explicit root within the top-level evaluation function. One-shot subcontinuations, like one-shot continuations [3, 23], are first-class but may be invoked at most once, a restriction obeyed by nearly all programs that use continuations [3].

Hieb et al. [17] describe subcontinuations in detail, give an operational semantics of a small language that incorporates subcontinuations, and describe a sequential implementation of subcontinuations. They also describe briefly how subcontinuations can be implemented in a concurrent setting, although until now a concurrent implementation has not been realized. The implementation requires that the control state of a concurrent computation be represented as a tree of stack segments, just as the control state of a sequential computation is represented as a stack of stack segments to support traditional continuations [18].

The implementation of one-shot subcontinuations described in this article uses threads to represent each stack segment in the tree of stack segments required by Hieb's implementation model. Using threads to represent stack segments has several advantages over incorporating support for subcontinuations into the lowest levels of a language implementation. In particular, it simplifies the implementation of both threads and continuations and provides a clear operational semantics for the interaction between them. It also provides, for the first time, a viable model for adding continuations to existing threaded implementations of other languages such as C and Java.

The remainder of this article is organized as follows. Section 2 discusses various continuation mechanisms and how they relate to threads. Section 3 describes subcontinuations in more detail and gives a few examples of their use. Section 4 describes the concurrent implementation of one-shot subcontinuations. This section identifies a small set of thread primitives and describes the implementation of subcontinuations in terms of these primitives. Section 4 also discusses how the implementation might be generalized to support multi-shot subcontinuations. Section 5 presents our conclusions.

2. Background

A continuation capture operation, using traditional continuations, creates an object that encapsulates the "rest of the computation." Invoking a traditional continuation discards (aborts) the entire current continuation and reinstates the previously captured continuation. Sometimes, however, finer control is required, i.e., only part of the continuation needs to be captured, or only part of the current continuation needs to be discarded when a continuation is reinstated. Felleisen [9, 10, 11] introduced the *prompt* operator to identify the base of a continuation and $\mathcal{F}$ to capture the continuation up to the last prompt. A continuation captured using $\mathcal{F}$ is *functional*, or *composable*, in that invoking it does not abort but rather returns to the current continuation. *Shift* and *reset* [6], which are based on a modified CPS transformation, are similar, differing primarily in that captured continuations include a prompt.

Subcontinuations generalize Felleisen's single prompt to multiple nested prompts and allow continuations to be used to control tree-structured concurrency [17]. In related work, Sitaram and Felleisen [27] show how nested prompts may be obtained from single prompts in a sequential setting. *Splitter* [26] extends the notion of a continuation in a manner similar to subcontinuations in a sequential setting but separates the continuation capture mechanism from the continuation abort mechanism. Gunter, et al. [13] describe how support for multiple prompts may be added to statically typed languages.

On a uniprocessor, both traditional and functional continuations are sufficient to implement multitasked threads at the source level [7, 15, 30]. Thus, many systems that support continuations provide no primitive support for threads. Continuations have also been used to implement threads on multiprocessors. MP [29] is a low-level interface designed to provide a portable multiprocessing platform. It provides an abstraction of a physical processor, operations to manage its state, and spin locks for mutual exclusion. Various concurrency abstractions, including threads, are implemented using first-class continuations on top of this interface.

Some systems provide native support for both threads and continuations. For example, Sting [20, 21] is a dialect of Scheme that provides general, low-level support for concurrency. Cooper et al. [5] describe a Mach-based multiprocessor threads implementation for Standard ML. Their package is based on the Modula-2+ threads package [2]. It includes mechanisms for mutual exclusion, synchronization and thread state. Multischeme supports both futures and continuations [25]. While each of these systems support both continuations and some form of threads, continuation operations are local to threads and cannot be used to control (abort or reinstate) groups of cooperating threads. Katz and Weise [22] also address the relationship between continuations and futures, but rather than providing a mechanism for controlling concurrency, they enforce a sequential semantics that makes concurrency transparent to the programmer.

One-shot continuations [3, 23] differ from ordinary multi-shot continuations in that a one-shot continuation may be invoked at most once. One-shot continuations can be implemented more efficiently than multi-shot continuations in stack-based implementations, because the stack segments representing a one-shot continuation need not be copied for later use when the continuation is reinstated. Most applications that use continuations use them in a one-shot manner. One-shot continuations cannot, however, be used to implement nondeterminism, as in Prolog [4], in which a continuation is invoked multiple times to yield additional values [8, 14]. This is the only application we have found that requires multi-shot continuations rather than one-shot continuations [3].

3. Subcontinuations

A subcontinuation [17] represents the rest of a subcomputation from a given point in the subcomputation. In the presence of tree-shaped concurrency, subcontinuations provide complete control over the process tree, allowing arbitrary nonlocal exits and reinstatement of captured subcomputations that may involve multiple threads.

The procedure *spawn* marks the *root* of a subcomputation and creates a *controller* that can be used to capture and abort the current subcontinuation up to and including the root. *spawn* takes a procedure *p* of one argument, creates a controller, and passes it to *p*. The controller can be invoked only in the dynamic extent of the procedure's invocation. If the controller is never invoked, then the value of the call to *spawn* is the value returned by *p*. Thus, the expression (*spawn* (**lambda** (*c*) (*cons* 1 2))) returns (1 . 2).

If the controller is applied to a procedure *q*, the subcontinuation from the point of controller invocation back to the root of the controller is captured, and *q* is applied to the captured subcontinuation in the continuation of the controller invocation. If the subcontinuation is never invoked, the effect is merely to abort the current subcomputation. The subcontinuation itself is non-aborting and is therefore composable. The subcontinuation captured includes the root of the controller. The root is reinstated on a subcontinuation invocation, allowing the controller to be invoked again. In the following simple example,

```
(cons 3
  (spawn (lambda (c)
           (cons 2
             (c (lambda (k)
                  (cons 1 (k '()))))))))
```

the call to *spawn* creates a controller *c* rooted within the *cons* of 3. Invoking this controller within the *cons* of 2 captures and aborts the continuation without disturbing the *cons* of 3; the captured continuation includes only the *cons* of 2. Invoking the captured continuation reinstates the *cons* of 2 within the *cons* of 1, so the value of the entire expression is (3 1 2).

The subcontinuation captured by a controller invocation can be invoked multiple times. In the following example, the controller is invoked in the base case of the factorial computation so that the subcontinuation *fact5∗* takes an argument and multiplies it by 120 (5!). So, the entire expression returns 14400 ($120 * 120 * 1$).

```
(define fact
  (lambda (n c)
    (if (= n 1)
        (c (lambda (k) k))
        (* n (fact (- n 1) c)))))

(let ((fact5* (spawn (lambda (c) (fact 5 c)))))
  (fact5* (fact5* 1)))
```

In the presence of concurrency, a subcontinuation captured and aborted by a controller invocation may encapsulate multiple threads of control. Invoking a subcontinuation that encapsulates multiple threads of control causes the concurrent subcomputation to resume. The *parallel-search* procedure (Figure 1) concurrently traverses a given tree looking for nodes that satisfy the specified predicate. On encountering such a node, it invokes the controller to suspend the search and returns the node along with a continuation that can be used to resume the search. We use **pcall** [1] here and in Section 4 to illustrate tree-structured concurrency, although any mechanism for introducing tree-structured concurrency would suffice, including a much more primitive *fork* operator. **pcall** evaluates its subexpressions in parallel and applies the procedural value of its first subexpression to the values of the remaining subexpressions. If none of the subexpressions involve side effects, then the **pcall** expression behaves like a normal procedure call.

Traditional continuations can be implemented in terms of subcontinua-

```
(define parallel-search
  (lambda (tree predicate?)
    (spawn
      (lambda (c)
        (letrec ((search
                   (lambda (tree)
                     (if (empty? tree)
                         #f
                         (pcall
                           (lambda (x y z) #f)
                           (if (predicate? (node tree))
                               (c (lambda (k)
                                    (cons (node tree) k)))
                           (search (left tree))
                           (search (right tree)))))))
          (search tree))))))
```

Figure 1: When a node satisfies *predicate?*, *parallel-search* invokes the controller *c* to suspend the search and returns a pair containing the node and a continuation that may be used to search for additional nodes.

tions by introducing an explicit root, via *spawn*, into the top-level evaluation function [17]. The traditional continuation operator, *call/cc*, is then defined in terms of the controller rooted in the top-level evaluation function. Although it is possible to implement *spawn* with *call/cc* in a sequential setting, doing so is less straightforward, as it involves the explicit simulation of the stack of stack segments required by a direct sequential implementation of subcontinuations.

One-shot subcontinuations are similar to one-shot continuations in that a captured one-shot subcontinuation may be invoked at most once. As with one-shot continuations, the stack segments representing a one-shot subcontinuation need not be copied when the subcontinuation is reinstated. In particular, as shown in Section 4.2, threads can be used to represent the stack segments required to implement one-shot subcontinuations without concern for restarting the same thread from the same point multiple times.

4. Implementation

This section describes the implementation of subcontinuations in terms of threads and is organized as follows. Section 4.1 describes a small set of

(*thread-fork thunk*)	forks a thread to invoke *thunk*.
(*thread-self*)	returns the current thread.
(*mutex-make*)	returns a new mutex.
(*mutex-acquire mutex*)	acquires *mutex*.
(*mutex-release mutex* [*thread*])	releases *mutex* (to *thread*, if specified).
(*condition-make mutex*)	returns a new condition associated with *mutex*.
(*condition-signal condition*)	signals *condition*.
(*condition-wait condition* [*thread*])	releases the mutex associated with *condition* (to *thread*, if specified) and waits for *condition* to be signaled, at which point the mutex is reacquired.
(*thread-block thread*)	blocks *thread*.
(*thread-unblock thread*)	unblocks *thread*.

Figure 2: Thread system features used to implement one-shot subcontinuations.

thread primitives that is sufficient for implementing subcontinuations. Section 4.2 presents the concurrent thread-based implementation of one-shot subcontinuations. Section 4.3 describes how multi-shot subcontinuations might be implemented using threads and discusses certain problems and restrictions.

4.1. Thread primitives

One-shot subcontinuations can be implemented in any language with a thread system powerful enough to support the set of thread-system features shown in Figure 2. In addition to the ability to dynamically fork threads, this set of features includes *mutexes* for mutual exclusion and *condition variables* [19, 24] for synchronization. It also includes primitives that allow a thread to block and unblock other threads. These features are supported at least indirectly by most modern thread systems, including the SGI IRIX thread system upon which our implementation is based.

A thread is created dynamically using *thread-fork*, which invokes its thunk argument in a separate thread. Threads are executed only for their effects.

Mutexes and condition variables provide a structured way of accessing shared resources. The order in which threads waiting on a mutex succeed in acquiring it is, on most systems, unspecified. In the implementation of subcontinuations, however, it is convenient to allow the thread releasing a mutex to specify the next thread that will succeed in acquiring the mutex. Thus, the primitives that release a mutex (*mutex-release* and *condition-wait*) take an optional second argument, which is the thread that will succeed in acquiring the mutex next. In the absence of direct thread system support for this feature, the equivalent functionality can be implemented by associating a "next thread" field with each mutex. When a thread successfully acquires a mutex with a nonempty next-thread field, the thread must check to see if it is indeed the next thread that is expected to hold the mutex. If it is not the designated thread it must release the mutex and wait again on it. Eventually, the specified thread will acquire the mutex.

The primitives *thread-block* and *thread-unblock* are required to control concurrent computations. They are asynchronous in that one thread can block or unblock a thread at any point in the other thread's execution. The operations themselves, however, must be synchronous in that they do not return until the specified thread is actually blocked or unblocked.

The threads interface described here is simple enough that it can be implemented on top of most existing thread packages. Mutexes and condition variables are supported by most systems[1]. Several systems (SGI IRIX, Linux, Solaris, POSIX, *etc.*) extend the UNIX signal mechanism to support threads. The signal mechanism can be used implement the thread blocking and unblocking procedures on those systems that do not support this functionality directly[2].

4.2. Subcontinuations from Threads

As described in Section 3, **pcall** provides a way to create tree-shaped concurrency, while *spawn* provides the ability to control tree-shaped concurrency. An example of tree-shaped concurrent computation is shown in Figure 3. Although **pcall** is not essential to the subcontinuation mechanism, its implementation is described here along with the implementation of *spawn* to illustrate how *spawn* interacts with **pcall**. The implementation of **pcall** is representative of the implementation of any operator used to introduce tree-structured concurrency.

[1]On systems that provide semaphores instead, mutexes and condition variables can be implemented in terms of semaphores.

[2]This requires the use of a user signal, e.g., SIGUSR1, rather than SIGSTOP, since a handler is needed to synchronize with the blocker before blocking.

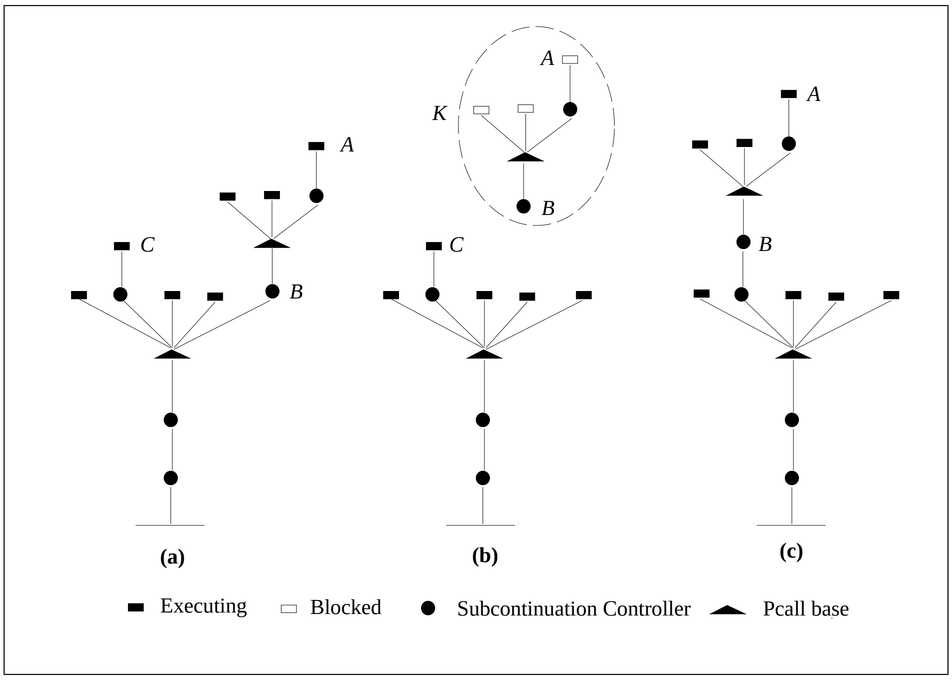

Figure 3: A process tree containing subcontinuation controller and pcall base nodes (a), subcontinuation capture by invoking the controller (b), and subcontinuation reinstatement by invoking the captured subcontinuation (c).

A **pcall** expression evaluates its subexpressions concurrently, then applies the value of its first expression to the values of the remaining expressions. The leaf in which the **pcall** is executed becomes a branch point, with a separate branch for each of the subexpressions. Thus, execution takes place only at the leaves of the tree. Once the subexpressions have been evaluated, the branch point becomes a leaf again and the procedure is applied to its arguments.

A call to *spawn* causes a subcontinuation controller to be inserted at the current execution point (Figure 3a). If control returns normally to that point, the controller is removed from the tree and execution is resumed. If a controller (say B) is invoked at one of the leaves (say A) in the subtree with B as the root, then that subtree is pruned and packaged into a subcontinuation K (Figure 3b). Subsequent invocation of that subcontinuation at a leaf (say C) causes the subcontinuation to be grafted onto the process tree at that leaf (Figure 3c).

(*make-index-list n*)	returns the list $(0\ 1\ 2\ \ldots\ n-1)$.
(*insert-pcall-node! threads*)	creates and inserts a **pcall** node at the current leaf.
(*insert-controller-node! thread*)	creates and inserts a controller node at the current leaf.
(*delete-child-node!*)	deletes the child of the current thread.
(*delete-pcall-edge! node edge*)	deletes *edge* from the given **pcall** node.
(*prune-subtree! node*)	prunes the subtree rooted at the given controller node.
(*graft-subtree! node*)	grafts the subtree rooted at the given controller node onto the tree at the current leaf.
(*controller-root node*)	returns the thread below the given controller node.
(*node->leaves node*)	returns a list of leaves in the subtree rooted at the given controller node.

Figure 4: Procedures used by **pcall** and *spawn* to maintain the subcontinuation data structures.

Subcontinuations are implemented by maintaining a tree of stack segments in which each stack segment is simply a stack of activation records. The key observation supporting the thread-based implementation of continuations is that a thread is, in essence, a stack of activation records. Thus, threads are used to represent stack segments. On subcontinuation capture, a subtree of threads is packaged into a subcontinuation after blocking the currently executing threads of the subcomputation (at the leaves of the subtree). When a subcontinuation is reinstated, the subtree of threads is grafted back onto the process tree and the computation is resumed by unblocking the threads at the leaves.

Because the unblocked threads may overwrite the activation records on their stacks, subcontinuations represented in this manner can be invoked at most once, i.e., they are one-shot subcontinuations.

The implementations of **pcall** and *spawn* share a common set of procedures that manipulate the data structures that make up the process tree. These procedures are listed in Figure 4. The code required to implement these procedures is straightforward and is not presented here.

All operations on the process tree must be atomic. Our implementation uses a single global mutex to serialize these operations. Most of the time, it is sufficient for the thread performing the operation to acquire the mutex, perform the operation, and release it. Some of the operations, however, require code to be executed by two different threads. To ensure the atomicity of the entire operation, the thread initiating the operation acquires the mutex, performs its half of the operation, and uses the optional next-thread parameter of the mutex-release procedure to pass the mutex to the second thread, which completes the operation before releasing the mutex.

4.2.1. Implementation of pcall

The code for **pcall** is shown in Figure 5, along with a help procedure, *pcall*∗. **pcall** itself is a syntactic extension. This syntactic extension simply makes thunks of the **pcall** form subexpressions, effectively delaying their evaluation, and passes them to *pcall*∗. When *pcall*∗ is invoked, it first obtains the mutex to gain exclusive access to the process tree. It then forks a thread for each of its arguments, updates the tree, and releases the mutex while it waits on the condition *done*, which is signaled when all the arguments have been computed. Each child thread concurrently computes one argument. Each result computed by a child is communicated to the parent thread via a specified slot in the vector *result*. The variable *counter* is decremented each time a child thread terminates to keep track of the number of arguments that have yet to be computed. When the last child finishes, the counter goes to zero and the child wakes up the parent thread by signaling the condition *done*. The parent then trims the tree, releases the mutex, and applies the resulting procedure to the resulting arguments.

4.2.2. Implementation of *spawn*

When *spawn* (Figure 6) is invoked, it obtains the process-tree mutex and creates a subcontinuation controller. It then forks a child thread, adds a node to the tree to mark the controller, and releases the mutex while it waits on the condition *done*.

The child thread applies the procedure *f* passed to *spawn* to the controller. Control can return from the call to *f* either by explicit invocation of the controller or by an ordinary return from *f*. The flag *controller-invoked?* is used to distinguish between these two cases.

The simpler case is when the controller is never invoked and the call to *f* returns *value*. In this case, the child thread enters the critical section, stores *value* in *result* to make it visible to the parent thread, and wakes up the parent by signaling the condition *done*. The child thread terminates and the parent thread resumes execution with *result* as the value returned by *spawn*.

```
(define-syntax pcall
  (syntax-rules ()
    ((_ proc arg ...)
     (pcall* (length '(proc arg ...))
        (lambda () proc)
        (lambda () arg) ...))))

(define mutex (mutex-make))

(define pcall*
  (lambda (n . args)
    (let ((result (make-vector n '*))
          (pcall-node '*)
          (done (condition-make mutex))
          (parent (thread-self))
          (counter n))
      (mutex-acquire mutex)
      (let ((thread-list
             (map (lambda (fn index)
                    (thread-fork
                      (lambda ()
                        (vector-set! result index (fn))
                        (mutex-acquire mutex)
                        (delete-pcall-edge! pcall-node index)
                        (set! counter (- counter 1))
                        (if (= counter 0)
                            (begin
                              (condition-signal done)
                              (mutex-release mutex parent))
                            (mutex-release mutex)))))
                  args
                  (make-index-list n))))
        (set! pcall-node (insert-pcall-node! thread-list)))
      (condition-wait done)
      (delete-child-node!)
      (mutex-release mutex)
      (let ((lst (vector->list result)))
        (apply (car lst) (cdr lst))))))
```

Figure 5: Implementation of **pcall**.

```
(define spawn
  (lambda (f)
    (define controller-node '*)
    (define controller-invoked? '*)
    (define controller-invocation-thunk '*)
    (define result '*)
    (define done (condition-make mutex))
    (define controller-wait
      (lambda ()
        (set! controller-invoked? #f)
        (condition-wait done)
        (if controller-invoked?
            (controller-invocation-thunk)
            (begin
              (delete-child-node!)
              (mutex-release mutex)
              result))))
    (define controller ;; See Figure 7
      (lambda (g) ...))
    (mutex-acquire mutex)
    (let ((thread
           (thread-fork
             (lambda ()
               (let ((value (f controller)))
                 (mutex-acquire mutex)
                 (let ((parent (controller-root controller-node)))
                   (set! result value)
                   (condition-signal done)
                   (mutex-release mutex parent)))))))
      (set! controller-node (insert-controller-node! thread))
      (controller-wait))))
```

Figure 6: Implementation of *spawn*.

```
(define controller
  (lambda (g)
    (mutex-acquire mutex)
    (let ((val '*)
          (continue (condition-make mutex))
          (root-thread (controller-root controller-node)))
      (set! controller-invocation-thunk
        (lambda ()
          (prune-subtree! controller-node)
          (let ((leaves (node->leaves controller-node)))
            (for-each thread-block leaves)
            (mutex-release mutex)
            (let ((k (lambda (v)
                       (mutex-acquire mutex)
                       (graft-subtree! controller-node)
                       (for-each thread-unblock leaves)
                       (set! val v)
                       (condition-signal continue)
                       (controller-wait))))
              (g k)))))
      (set! controller-invoked? #t)
      (condition-signal done)
      (condition-wait continue root-thread)
      (mutex-release mutex)
      val)))
```

Figure 7: Implementation of *controller*.

The more complicated case is when the controller (Figure 7) is actually invoked at one of the leaves. When this happens, the subcontinuation that represents the subtree rooted at the controller (Figure 3b) must be captured and aborted. To do this, the mutex is acquired to ensure that no other thread starts a continuation operation on the process tree. The thread that invokes the controller determines the thread *root-thread* that is waiting at the subcontinuation controller, packages the work to be done by that thread into a thunk *controller-invocation-thunk*, updates the variable *controller-invoked?* to inform it that the controller was invoked, and wakes up the controller by signaling the condition *done*. It then waits on condition *continue* while handing the mutex to *root-thread*.

When *root-thread* starts executing, the *controller-invoked?* flag indicates that the controller was called explicitly, so it invokes *controller-invocation-thunk* to capture the subcontinuation. This causes the tree rooted at the controller to be pruned and the threads executing at its leaves to be blocked. Then, after leaving the critical section, it creates a subcontinuation k and applies the controller argument g to this subcontinuation.

Later, if the subcontinuation k is invoked by a thread executing at a leaf, the thread obtains the process-tree mutex, grafts the process subtree captured as part of the subcontinuation onto the current leaf, and unblocks all the threads at the leaves of the grafted subtree (Figure 3c). It then stores the value v to be returned to the point where the controller was invoked in *val*, signals the condition *continue* to the thread that invoked the controller, and waits for the condition *done*. The signaled thread returns the value now stored in *val* to the point where the controller was invoked.

To avoid deadlock, a thread is never blocked while holding the mutex. To maintain this invariant, the blocking thread always holds the mutex until after the (synchronous) blocking operation has succeeded.

4.3. Multi-shot subcontinuations

The implementation described in the preceding section can be extended to support multi-shot subcontinuations. This requires a thread cloning operator *thread-dup* that allows the threads captured in a subcontinuation to be cloned. The clones are used to restart the subcomputation captured in the subcontinuation, while the original ones are kept around for subsequent invocations of the subcontinuation.

Three complications arise in this method for extending the implementation to support multi-shot subcontinuation. First, a single controller or **pcall** node can now exist at multiple locations in the process tree. Therefore, the data structures used to communicate between parent threads and their children must be moved from the procedures where they are currently encapsulated into the process-tree data structure, and they must be cloned whenever the corresponding node in the tree is cloned. At the start of each operation, the correct node in the tree must be located and the corresponding data structures used.

Second, condition variables are also used on a per-node basis. Each condition variable has at most one thread waiting on it, and that thread is awakened by signaling that condition. Simple replication of the condition variables, however, in order to retain the property that only one thread can wait on a condition variable at a time, does not work. This is because some of the threads are waiting on the condition variables when they are cloned, which therefore cannot be replaced. One solution to this problem is to accept the fact that multiple threads might be waiting on a condition variable and wake all of them up using a *condition-broadcast* primitive. Each thread, when awakened, would have to check to see if it is the intended target of the wakeup message. If it is not, it must again wait on the condition variable.

Third, compiler support is required to allow the stack encapsulated within a thread to be copied. In particular, mutable variables and data structures must be stored outside of the stack or accessed from the original stack via an extra level of indirection. In contrast, no compiler support is required to implement one-shot subcontinuations.

5. Conclusions

It has long been known that continuations can be used to implement threads. In this article, we have shown that threads can be used to implement continuations. In so doing, we have provided the first detailed implementation model for subcontinuations in a concurrent setting, furthered the understanding of the relationship between continuations and threads, and provided a straightforward operational semantics for the interaction between threads and subcontinuations.

We have implemented a complete thread package that supports the primitives described in Section 4.1 as part of *Chez Scheme* on an SGI *Power Challenge* multiprocessor and used this package to implement one-shot subcontinuations (available via `http://www.cs.princeton.edu/~skumar/subK`). We have not yet extended the implementation with support for multi-shot subcontinuations.

Although the subcontinuation mechanism as described uses higher-order procedures, they are not essential to the mechanism or to its implementation. Because the implementation of one-shot subcontinuations does not rely upon higher-order procedures or on any special compilation techniques, the model is applicable to any language or language implementation that supports the small set of thread primitives described in Section 4.1. Thus, the model demonstrates for the first time a straightforward way to add support for first-class continuations to threaded versions of other languages, such as C and Java, without changes to the language or compiler.

In languages without automatic storage management, such as C, the programmer must be responsible for deallocating unused subcontinuations and any data structures used only by the threads that represent the subcontinuations. This is nothing new: programmers in such languages already face a similar problem when threads are killed and when computations are aborted via `longjmp` or other nonlocal exits.

The techniques used to implement one-shot subcontinuations can be adapted to support one-shot variants of the other continuation mechanisms described in Section 2, including *call/cc*. Multi-shot variants of these other mechanisms can be supported as well, subject to the complications discussed in Section 4.3. In the absence of concurrency, the asynchronous thread operators would not be needed, since all but the one active thread would be blocked waiting on condition variables.

Acknowledgements: Comments by Oscar Waddell and anonymous reviewers led to substantial improvements in the presentation of this material.

References

1. Halstead, Jr., Robert H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985) 501–538.

2. Birrell, Andrew, Guttag, John V., Horning, James J., and Levin, Roy. Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proceedings of the Eleventh Symposium on Operating Systems Principles* (November 1987) 94–102.

3. Bruggeman, Carl, Waddell, Oscar, and Dybvig, R. Kent. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation* (May 1996) 99–107.

4. Clocksin, William F. and Mellish, Christopher S. *Programming in Prolog*. Springer-Verlag, second edition (1984).

5. Cooper, Eric C. and Morrisett, J. Gregory. *Adding Threads to Standard ML*. Technical Report CMU-CS-90-186, Computer Science Department, Carnegie Mellon University (December 1990).

6. Danvy, Olivier and Filinski, Andrzej. Representing control: A study of CPS transformation. *Mathematical Structures in Computer Science*, 2, 4 (1992) 361–391.

7. Dybvig, R. Kent and Hieb, Robert. Engines from continuations. *Computer Languages*, 14, 2 (1989) 109–123.

8. Felleisen, Matthias. *Transliterating Prolog into Scheme*. Technical Report 182, Indiana University (October 1985).

9. Felleisen, Matthias. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (January 1988) 180–190.

10. Felleisen, Matthias, Friedman, Daniel P., Duba, Bruce, and Merrill, John. *Beyond Continuations.* Technical Report 216, Indiana University Computer Science Department (1987).

11. Felleisen, Matthias, Wand, Mitchell, Friedman, Daniel P., and Duba, Bruce F. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (July 1988) 52–62.

12. Friedman, Daniel P., Haynes, Christopher T., and Wand, Mitchell. Obtaining coroutines with continuations. *Computer Languages*, 11, 3/4 (1986) 143–153.

13. Gunter, C. A., Rémy, R., and Riecke, Jon G. A generalization of exceptions and control in ML-like languages. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture* (June 1995).

14. Haynes, Christopher T. Logic continuations. *LISP Pointers* (1987) 157–176.

15. Haynes, Christopher T. and Friedman, Daniel P. Engines build process abstractions. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming* (August 1984) 18–24.

16. Hieb, Robert and Dybvig, R. Kent. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1990).

17. Hieb, Robert, Dybvig, R. Kent, and Anderson, III, Claude W. Subcontinuations. *Lisp and Symbolic Computation*, 7, 1 (March 1994) 83–110.

18. Hieb, Robert, Dybvig, R. Kent, and Bruggeman, Carl. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (June 1990) 66–77.

19. Hoare, C.A.R. Monitors: An operating system structuring concept. *Communications of the ACM*, 17, 10 (1974) 549–557.

20. Jagannathan, Suresh and Philbin, James. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1992).

21. Jagannathan, Suresh and Philbin, James. A foundation for an efficient multi-threaded Scheme system. In *Proceedings of the 1992 Conference on Lisp and Functional Programming* (June 1992).

22. Katz, Morry and Weise, Daniel. Continuing into the future: On the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (June 1990) 176–184.

23. Komiya, Tsuneyasu and Yuasa, Taiichi. Indefinite one-time continuation. *Transactions Information Processing Society of Japan*, 37, 1 (1996) 92–100.

24. Lampson, Butler W. and Redell, David D. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23, 2 (1980) 105–117.

25. Miller, James S. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology (September 1987).

26. Queinnec, Christian and Serpette, Bernard. A dynamic extent control operator for partial continuations. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages* (January 1991) 174–184.

27. Sitaram, Dorai and Felleisen, Matthias. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3, 1 (January 1990) 67–99.

28. Sussman, Gerald J. and Steele Jr., Guy L. *Scheme: An Interpreter for Extended Lambda Calculus.* AI Memo 349, Massachusetts Institute of Technology Artificial Intelligence Lab (1975).

29. Tolmach, Andrew and Morrisett, J. Gregory. *A Portable Multiprocessor Interface for Standard ML of New Jersey.* Technical Report TR-376-92, Princeton University (June 1992).

30. Wand, Mitchell. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference* (August 1980) 19–28.